Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2323689 A1 2001/12/02

(21) 2 323 689

(12) DEMANDE DE BREVET CANADIEN
CANADIAN PATENT APPLICATION

(13) A1

(22) Date de dépôt/Filing Date: 2000/10/17
(41) Mise à la disp. pub./Open to Public Insp.: 2001/12/02
(30) Priorité/Priority: 2000/06/02 (2,310,943) CA

(51) Cl.Int.$^7$/Int.Cl.$^7$ H04L 29/02, G06F 17/60

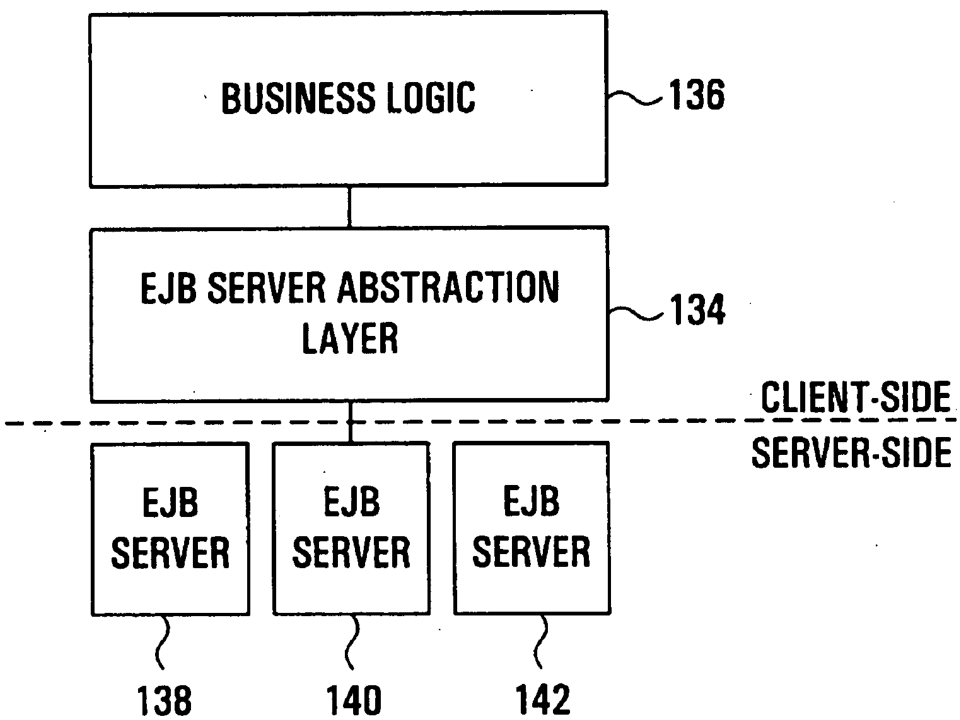
(71) Demandeur/Applicant:
SERVIDIUM INC., CA

(72) Inventeurs/Inventors:
SIKORSKY, MICHAEL J., CA;
SHAW, ROBERT, CA;
RASMUSSON, JONATHAN, CA

(74) Agent: SMART & BIGGAR

(54) Titre : TECHNIQUES, LOGICIELS, OBJETS ET METHODES D'ABSTRACTION DE SERVEUR DE STRUCTURES ORIENTE OBJETS REPARTIS

(54) Title: DISTRIBUTED OBJECT ORIENTED STRUCTURE SERVER ABSTRACTION TECHNIQUES, SOFTWARE, OBJECTS AND METHODS




(57) Abrégé/Abstract:
A software abstraction layer abstracts away differences among different Enterprise JavaBean (EJB) servers. The software abstraction layer has an abstract class and a number of concrete or sub-classes of the abstract class. Each concrete class is associated with a particular EJB server. The EJB servers offer a number of the same or common services. The abstract class includes an abstract method for each of the common services. Each concrete class includes a concrete method associated with each of the abstract methods. Each concrete method implements one of the common services on a particular EJB server.

78746-4

## Abstract of the Disclosure

A software abstraction layer abstracts away differences among different Enterprise JavaBean (EJB) servers. The software abstraction layer has an abstract class and a number of concrete or sub-classes of the abstract class. Each concrete class is associated with a particular EJB server. The EJB servers offer a number of the same or common services. The abstract class includes an abstract method for each of the common services. Each concrete class includes a concrete method associated with each of the abstract methods. Each concrete method implements one of the common services on a particular EJB server.

78746-4

# Distributed Object Oriented Structure Server Abstraction Technique, Software, Objects and Methods

## Field of the invention

The invention relates to software abstraction and is particularly concerned with abstracting away differences among different distributed object oriented structure servers such as, for example, Enterprise JavaBean™ (EJB™) servers.

## Background of the invention

Enterprise JavaBeans™ (EJBs™) are server-side software components, written in /the Java programming language and used within the Java™ 2 Enterprise Edition (J2EE™) platform. EJBs are distributed objects which provide remote services for clients distributed on a network. An EJB represents a business concept like a customer or a hotel clerk which is capable of performing a set of processes or tasks. For example, an EJB representing a customer may perform tasks such as determining the customer's bank balance or adding to the customer's bank balance.

To create an EJB, an EJB developer provides two interfaces, a home interface and a remote interface, that define an EJB's business methods, plus the actual EJB implementation class. EJBs are accessed by client applications over a network through their remote and home interfaces. The remote and home interfaces expose the capabilities of the EJB and provide all the methods needed to create, update, interact with, and delete the EJB.

EJBs run in an environment called an EJB container. The EJB container is responsible for managing EJB components and running them. The EJB container manages aspects of an EJB component at runtime including remote access, security,

persistence, transactions, concurrency, and providing access to and pooling of resources.

The EJB container isolates the EJB component from direct access by client applications. Accordingly, the EJB developer can focus on encapsulating business rules, while the container takes care of everything else.

An EJB server provides the runtime environment for one or more EJB containers. EJB servers manage the low-level system resources required by the EJB containers. An EJB server may contain one or more EJB containers.

The widespread adoption of the Java language as an enterprise application development language has lead to a large offering of commercial products from different vendors. Because of the immaturity of Java technology and the EJB specification, many vendors' development tools sold to software developers provide proprietary solutions. These proprietary services aid the developer at the expense of locking the developer's EJB-related applications to a particular vendor's tool. This "vendor lock-in" is counter to the open spirit of Java development.

With the proliferation of commercial EJB servers in the market place, the likelihood of developers at some point migrating (or wanting to migrate) from one vendor server to another increases. Migrating EJBs from one EJB server to another can be difficult.

Lack of clarity in the EJB specification has forced developers to implement certain server features in their own proprietary way. Accordingly, there are often subtle implementation differences between EJB servers offered by different vendors. As a result, considerable code rewriting is

typically required to modify EJBs and their respective applications to facilitate migration from one EJB server to another.

Vendor lock-in manifests itself in software by having EJBs developed that only run within a single vendor's runtime environment. Hence, a developer's application is dependent on a particular vendor's EJB server implementation.

Such vendor lock-in ties an application to a particular vendor making the application extremely vulnerable to that vendor. For instance, if a vendor decided to stop supporting certain features of its EJB server, any developers using that vendor's EJB server would be forced to use an unsupported version of the vendor's EJB server. If a client application requires new features offered by a newer version of a vendor's EJB server, then considerable effort is typically required to rewrite the developer's application in such away that it is no longer dependent on the proprietary older features.

Further, many developers give little thought regarding how to best implement and architect their EJB design in anticipation of moving an application from one EJB server to another. Hence EJB applications are often developed without much thought on how to avoid vendor lock-in. The result is typically an expensive, time consuming, application migration, if it is decided to use a different EJB server.

## Summary of the invention

According to a first aspect, the invention provides a computer-readable medium containing executable instructions for abstracting away differences among two or more distributed object oriented structure (DOOS) servers, the instructions

being adapted to be used in combination with business logic software code for communicating with a DOOS, a DOOS having a home interface, a remote interface and an implementation class, wherein the instructions when loaded in a computer, adapt the computer to receive the identity of a current DOOS server from the business logic software code, and enable the business logic software to communicate with a DOOS moved from a previous DOOS server to a current DOOS server.

According to another aspect, the invention provides a computer-readable medium storing computer software and data that when loaded by a computing device define object oriented objects for use in abstracting away differences among two or more distributed object oriented structure (DOOS) servers, the two or more DOOS servers supporting a number of common service, a DOOS having a home interface, a remote interface, and an implementation class, the computer software and data being implementation by the computing device in an object oriented framework comprising: a first object comprising an abstract class having, for each of the common services, a corresponding abstract method; and for each of the DOOS servers, an associated object comprising a concrete sub-class of the abstract class of the first object, each concrete sub-class comprising, for each of the abstract methods of the abstract class, an associated concrete method, wherein each concrete method is adapted to implement a service on the associated DOOS server.

According to another aspect, the invention provides a computer implemented method for abstracting away differences among two or more distributed object oriented structure (DOOS) servers, a DOOS having a home interface, a remote interface and an implementation class, the DOOS servers supporting a number of common services, the method comprising implementing business logic and a DOOS server abstraction layer wherein: the business

logic: (a) provides an identity of a DOOS server incorporating one or more DOOSs; (b) obtains, through the DOOS server abstraction layer, for each of the one or more DOOSs, an associated home interface; (c) obtains, for each of the one or more DOOSs, an associated remote interface; (d) for each DOOS, uses the associated home interface and the associated remote interface to communicate with the DOOS through the DOOS abstraction layer, using non-DOOS server specific abstract methods; the DOOS server abstraction layer: (e) obtains the home interface for each of the one or more DOOSs for the business logic; and (f) associates each non-DOOS server specific abstract method with a DOOS server specific concrete method.

Advantageously, different embodiments of the present invention may permit:

Providing easier migration of EJB applications from one EJB server to another, thereby reducing time, effort and expense;

Reducing the dependence of an EJB developer upon a particular EJB server;

Reducing the dependence of an EJB developer upon a particular EJB server vendor.

## Brief Description of the Drawings

Preferred embodiments of the invention will now be described with reference to the attached drawings in which:

FIG 1 illustrates multiple network interconnected devices;

FIG 2 is a block diagram of the organization of computer memory of the devices of FIG 1;

FIG 3 is a schematic diagram providing an example of the relationship between EJBs, EJB containers and an EJB server;

FIG 4 is a high level block diagram illustrating an EJB software abstraction layer inserted between business logic software code and particular EJB servers, according to an embodiment of the present invention;

FIG 5 is a high level block diagram showing inputs to and outputs of a system according to the embodiment of FIG 4;

FIG 6 is a high level block diagram illustrating mechanisms incorporated within the business logic and the EJB server abstraction layer, in accordance with an embodiment of the present invention;

FIG 7 is a conceptual diagram of object oriented structures associated with the business logic and the EJB server abstraction layer, in accordance with an embodiment of the present invention;

FIG 8 is a high level block diagram showing conceptual groupings of object oriented structures used in an example of an embodiment of the present invention;

FIG 9 is a sequence diagram illustrating a series of steps implemented with respect to the example of FIG 8;

FIG 10 is a high level block diagram illustrating an example of a method for obtaining the type of EJB server to be used according to another embodiment of the present invention;

FIG 11 is a sequence diagram depicting an order of events for the example of FIG 10.

Detailed Description

FIG 1 illustrates, by way of example, digital computer network 100 including computing devices 104, 106 and 108. Devices 104, 106 and 108 may, by way of example, be personal computers; mainframe computers; networking equipment such as routers, switches, frame relays; telephone switching equipment; or the like. Network 100 may for example, be a wide area network, conforming to any of a number of known networking protocols including TCP/IP, IPX, Appletalk, UUCP or the like. Alternatively, network 100 may be a local area network; a collection of interconnected smaller computer networks comprising an intranet or internet; or a portion of the public internet. As will be appreciated and as illustrated, network 100 may be interconnected with other networks such as the public internet.

Computing devices 104, 106 and 108 are all network management devices that store network management data, among other things. Management of any of the network devices may be accomplished by access to one device 104, 106 or 108 by one of the other devices 104, 106 or 108, or other interconnected devices (not shown), in communication with network 100.

Each computing device 104, 106, and 108 comprises a processor interconnected with storage memory and a network interface. Additionally, each computing device may comprise an input/output peripheral capable of reading data from a computer readable storage medium, such as a floppy diskette, CD-ROM, tape or the like. The organization of memory of computing device 104 is illustrated in FIG 2. The organization of memory of computing devices 106, and 108 is similar, but not illustrated. As shown, loaded within memory of each computing device is operational software, including preferably an operating system 112; network interface software 114; and application software 116. As will be appreciated, the

operational software may be loaded from computer readable storage medium 110.

Operating system 112, may for example be a UNIX operating system. Network interface software 114 typically comprises software allowing communication of device 104 with the remainder of network 100 using a known network protocol. Network interface software 114 may, for example, be an internet protocol suite, and could optionally form part of operating system 112. Application software 116 typically comprises software that in combination with operating system 112, provides other desired functionality of devices 104, 106 and 108.

In a distributed system, certain software and/or hardware is considered as server-side components and other software and/or hardware is considered as client-side components. For the purpose of the following description and examples, the client-side components will be considered to be resident on the device 106 and the server-side components will be considered to be resident on device 104. It is also possible that both the server-side components and the client-side components could be located on the same device 104, 106 or 108.

Enterprise JavaBeans (EJBs) are the result of a compiled executing program or groups of programs written in the Java programming language. EJBs are distributed object oriented structures (DOOS), formed as a result of object oriented classes and interfaces described below. Each EJB is an object oriented class, that when compiled and executed in memory of the device 104, forms an instance of an object having attributes and functions. FIG 3 is a schematic diagram providing an example of the relationship between EJBs 120-126, EJB containers 128, 130 and an EJB Server 132, all of which are typically considered as software components.

An EJB server 132 may contain any number of EJB containers. An EJB container 128, 130 may contain any number of EJBs. In this example, the EJB server 132 contains two EJB containers, namely EJB container 128 and EJB container 130; the EJB container 128 contains two EJBs namely EJB 120 and EJB 122; and the EJB container 130 contains two EJBs, namely EJB 124 and EJB 126.

FIG 4 is a high level block diagram illustrating an EJB server abstraction layer 134 inserted between business logic software code 136 (described below) and different EJB servers EJB server 138, EJB server 140 and EJB server 142. The EJB abstraction layer 134 and the business logic software code 136 are both client-side applications or components, which may be resident as part of the application software 116 of computing device 106 (FIGs 1 and 2). The EJB servers 138, 140, 142 are server-side applications or components, along with their associated EJB containers 128, 130 and EJBs 120-126, which may be resident as part of the application software 116 of computing device 104 (FIGs 1 and 2).

The EJB servers 138, 140 and 142 each represent a separate EJB server, each possibly being sold by a different vendor. The EJB server abstraction layer 134 is software code that acts as an interface or layer between the business logic 136 and the particular EJB servers 138, 140 and 142. As a result, the EJB server abstraction layer 134 allows the business logic 136 to remain the same, regardless of which server 138, 140 or 142 is being used.

The business logic 136 is typically a set of Java classes and Java objects that solve a business problem. In object oriented programming terms, the business logic 136 is decomposed into a set of objects and components. The business logic 136 defines the structure and nature of business objects.

The business logic 136 also determines the external behavior of how objects (not shown) defined within the business logic 136 interact with other objects in a system 144 (FIG 5). The system 144 is a Java language mechanism for dynamically binding objects to abstract classes and interfaces at runtime. The system 144 may be thought of as a Java Virtual Machine.

Another way to conceptualize the block diagram of FIG 4 is shown in the block diagram of FIG 5, showing inputs and the output to the system 144. In general terms, the inputs to the system 144 are the components that can change the system 144. In the example shown in FIG 5, there are three potential inputs, each being a different vendor EJB server, EJB server 138, EJB server 140 or EJB server 142. Only one EJB server 138, 140 or 142 at a time can be an input to the system 144.

The system 144 is an implementation of the object oriented programming concept of polymorphism. In other words, the output, the EJB server abstraction layer 134, is the abstraction of the input, namely one of the EJB servers 138, 140 or 142. As a result, the output, the EJB server abstraction layer 134, is polymorphic in nature, which means that it can take many "shapes". In other words, the output, the EJB server abstraction layer 134, could represent any one of three inputs 138-142, in abstract terms. As a result, any client using the output, the EJB server abstraction layer 134, would not care what the input 138, 140 or 142 actually was. The client can use the output, the EJB server abstraction layer 134, in the same manner, regardless of the whether the input to the system 144 was EJB server 138, EJB server 140 or EJB server 142.

EJBs rely upon other features or components of the J2EE platform such as the Java Messaging Service™ (JMS™) and the Java Naming and Directory Interface™ (JNDI™). JNDI

provides Java platform-based applications with a unified interface to multiple naming and directory services. JMS is a set of object services for handling store-and-forward messaging requirements.

EJB servers from different vendors may use JMS and/or JNDI, for example, in different ways. The following example illustrates how the EJB server abstraction layer 134 abstracts away different methods by which different EJB server vendors use JMS and JNDI so that the business logic 136 need not concern itself about which EJB server is being used. However, other J2EE technologies or features such as, for example, the Java Database Connectivity™ (JDBC™) and Java Transaction API and Service™ (JTS/JTS™) features could be abstracted away in the same manner.

Each of the business logic 136 and the EJB server abstraction layer 134 has a number of mechanisms for communicating with EJBs. Each EJB, as noted in greater detail below, has three parts (see FIG 8), namely a remote interface 170, a home interface 172 and an implementation of the EJB itself 174.

Referring to the block diagram of FIG 6, the business logic 136 has a number of mechanisms (classes, objects, methods, etc.) including the following:

-a mechanism 150 for solving a business problem using one or more EJBs;

-a mechanism 152 for identifying a particular EJB server to be used;

-a mechanism 154 for communicating with the EJB server abstraction layer 134 to obtain home interfaces for the one or more EJBs;

-a mechanism 156 for obtaining remote interfaces for the one or more EJBs;

-a mechanism 158 to communicate with the one or more EJBs, through the EJB abstraction layer 134, using abstract or non-EJB server specific method calls.

The EJB server abstraction layer 134 has mechanisms 160-163 for allowing the business logic 136 to communicate with a specific EJB server without consideration of which EJB server is being used. The EJB server abstraction layer 134 has a mechanism 160 for locating the one or more EJBs, by which it determines the home interfaces for the one or more EJBs. The mechanism 160 then returns the home interfaces for the one or more EJBs. The EJB server abstraction layer 134 also provides abstract or non-EJB server specific method calls 162. The EJB server abstraction layer 134 also provides a mechanism 163, where, for each EJB server, a concrete method is associated with each abstract or non-EJB server specific method. Each concrete method is adapted to implement a specific service on a specific EJB server.

FIG 7 illustrates a conceptual view of objects 164-167 which could be used, at least in part, to implement mechanisms 162, 163 of FIG 6. The mechanism 162 of FIG 6 provides for the use of an abstract server object 164 encapsulating abstract method1, abstract method2, etc, as shown in FIG 7. Mechanism 163 of FIG 6 provides that, for each EJB server, a concrete method is associated with each abstract or non-EJB server specific method. FIG 7 provides a high level view of this concept.

For example, referring to FIG 7, the abstract EJB server object 164 has a particular abstract method1 for performing a particular service. Each EJB server 138, 140 and

142 is able to provide that service. However, each EJB server 138, 140 and 142 may require different input parameters to perform the service. As shown in FIG 7, the abstract EJBserver object 164 provides abstract method1 which has a corresponding concrete method1 in each of the concrete EJB server objects 165, 166 and 167. The concrete method1 within the concrete EJB server object 165 could be the same as or different from the concrete method1 within the concrete EJB server object 166, for example, depending upon whether or not their corresponding EJB servers 138 and 140 require the same or different inputs for this particular method.

Example

The following example is a bank metaphor.

FIG 8 is a high level block diagram showing conceptual groupings used in the example. The business logic 136 and the EJB server abstraction layer 134 represent client-side code. The actual EJB server 132 represents server-side code.

The example simulates a simple transaction of a bank teller 169, making a deposit into a customer bank account. The example includes the business logic 136, the EJB server abstraction layer 134, and an EJB, namely a CustomerEJB 120 residing within an EJB container 128. The client classes consist of two primary groups, namely the business logic 136 and the EJB abstraction layer 134. The business logic 136 contains the logic used to build the application, and would be written by a developer when solving a business problem. The EJB abstraction layer 134 is infrastructure code that would be used by the developer to communicate indirectly with the actual EJBServer 132 and the CustomerEJB 120. The server side classes

show the Customer EJB 120 that resides within the EJB container 128.

All EJBs consist of three parts:

A Remote interface, which in this example is referred to as CustomerRemote interface 170. This is an interface used by a client 176 to communicate with the CustomerEJB object 120 within the EJB container 128;

A Home interface, which in this example is referred to as CustomerHome interface 172. This is an interface used by the client 176 to create the remote interface, CustomerRemote interface 170; and

- An EJB, which in this example is referred to as CustomerBean 174. This is an implementation class which is the implementation of the CustomerEJB 120 itself.

For this example, it will be assumed that the CustomerEJB object 120 has been created and deployed to the actual EJB server 132. The type of EJB server (JBoss™, WebLogic™, iPlanet™, etc) does not matter.

The following is a brief description of each of the classes and objects used in the example:

Within the business logic 136:

- BankTeller object 169 is used to make a deposit into a customer's bank account.
- CustomerMgr object 178 looks up and obtains a reference to a customer.
- Customer object 182 represents a customer at a bank.

Within the EJB server abstraction layer 134:

- Abstract EJBServer object 184 provides an abstract view of any vendor's EJB server, whether it is a JBossServer object 186, a WebLogicServer object 188, an iPlanetServer object 190, or any other EJB server object (not shown).
- JBossServer object 186 is a JBoss concrete implementation of an abstract EJB server object 184.
- WebLogicServer object 188 is a WebLogicServer concrete implementation of an abstract EJB server object 184.
- iPlanetServer object 190 is an iPlanetServer concrete implementation of an abstract EJB server object 184.

Within the CustomerEJB object 120

- CustomerRemote interface 170 is a remote interface used by the client 176 to communicate with the CustomerEJB object 120.
- CustomerHome interface 172 is the home interface used by the client to look up and obtain a reference to a customer.
- CustomerBean 174 is the implementation of the EJB itself.

Sample Code

Sample Java code for implementing this example is set out at appendix "A" to the detailed description, hereinafter referred to as the "Sample Code". However, the Sample Code is not complete code. For example, exception handling, import statements, etc have not been included for clarity. Any classes in the Sample Code that are not referred to in the

preceding paragraph are either part of the standard Java programming language or are part of the J2EE specification, and would automatically be generated by the actual EJB Server 132. For example, the InitialContext object referred to in the Sample Code is a JNDI object that would be created by the actual EJB server 132.

The Sample Code demonstrates how the business logic code 136 can interact with the EJB server abstraction layer 134 by changing a single line of code in the business logic code 136 if the actual EJB server 132 changes. A second example, described below, according to another embodiment, demonstrates how the business logic software code 136 need not change at all even if the actual EJB server 132 changes.

As noted above, in the Sample Code, only a single line of code in the business logic 136 needs to be changed if the actual EJB server 132 changes. The single line of code in the business logic 136 to be changed appears in the following portion of the Sample Code:

```
// CUSTOMER MANAGER
public class CustomerMgr
{
   EJBServer server = null;

   public CustomerMgr()
   {
      // get our EJB server our application is using today
      server = new JBossServer();
      //server = new WebLogicServer();
      //server = new iPlanetServer();
   }

...
}
```

In the code set out above, the server object "server" is an abstract interface that shields the business logic 136 from the underlying EJB details regardless of which actual EJB server 132 is used. The CustomerMgr class knows that it has an actual EJB server 132, and that it can perform a number of

requests on the server 132. However, the CustomerMgr object 178 does not know which concrete implementation of an actual EJB Server it is using. In other words, in this example, the CustomerMgr object 178 does not know if the actual EJB server 132 is actually a Jboss Server, a WebLogic Server or an iPlanet Server.

In the Java language, comments are shown by "//", which means that everything to the right of "//" is ignored. Therefore, in the code set out above, the line "server = new JBosServer();" is implemented and the two following lines are ignored, because they have been commented out.

When the CustomerMgr object 178 actually instantiates an instance of an EJBServer abstract class:

"server = new JBossServer();"

it now has a concrete implementation to which it can delegate calls. When the client 176 needs to move to another actual EJB server 132, it changes the concrete implementation. For example, if the actual EJB server 132 were to change from a JBoss server to a WebLogic server, the line:

"server = new JBossSever();"

would be commented out and the line:

"server = new WebLogicServer(); "

would be implemented by deleting "//" so that it is no longer commented out. The CustomerMgr object 178 can now delegate calls to the WebLogic server.

The Sample Code shows that an abstract type EJBServer is declared. The Sample Code also shows that each of the concrete sub-classes JBossServer 186, WebLogicServer 188 and

iPlanetServer 190 is defined by extending the abstract class EJBServer. Accordingly, details regarding which concrete server is being used are hidden.

The EJB server abstraction layer 134, as set out in the abstract class EJBServer in the Sample Code, shows all the methods that are supported by the concrete implementations of the abstract EJBServer object 184. EJBServer abstract methods offer no details describing how these methods are implemented. This is left to the concrete sub-classes JBossServer 186, WebLogicServer 188 and iPlanetServer 190. At runtime, it is the EJB server abstraction layer 134 that the business logic 136 uses when making calls to the EJB server objects 186, 188, 190. Details regarding which EJB server object is used are hidden from the business logic 136.

As noted above, the concrete implementations of the EJB servers, namely the server objects JbosServer 186, WebLogicServer 188 and iPlanetServer 190 perform the work involved to realize the client requests. It is in the EJB server abstraction layer 134 that the details regarding how to interact with a particular EJB server object 186-190 are implemented. For instance, the GetInitialContext method for the JBossServer class is different from that in the WebLogicServer class. In other words, the GetInitialContext method for the JBossServer class uses a different lookup string to obtain the JNDI InitialContext object, than the WebLogicServer class (ie: "org.jnp.interfaces.NamingContextFactory" instead of "weblogic.jndi.WLInitialContextFactory"). In both cases, as shown in the Sample Code, the InitialContext object is used to look up an EJB home factory. The business logic 136 need not be aware of these differences.

The Sample Code will now be considered with reference to the sequence diagram of FIG 9. Each of the boxes near the top of the sequence diagram of FIG 9 represents an object.

- At step 200, the main() method of the BankTeller class is implemented. The BankTeller object 169 creates a reference to itself and calls the deposit method.

- At step 202, the BankTeller object 169 creates the CustomerMgr object 178.

- At step 204, in the constructor of the CustomerMgr object 178, a reference to the abstract EJBServer object 184 is created. In this example, the abstract EJBServer object 184 instantiated is of type JBossServer 186.

- At step 206, the BankTeller object 169 now asks the CustomerMgr object 178 to find a customer named "Peter".

- At step 208, because customer Peter is realized as an EJB on the actual EJB server 132, the CustomerMgr object 178 must use the abstract EJBServer object 184 it created to obtain a reference to the CustomerEJB 120.

- At step 210, the JBossServer object 186 obtains an InitialContext object relating to its EJB container 128. The InitialContext object is used to look up the EJB home interface, CustomerHome interface 172.

- At step 212, the JBossServer object 186 asks the InitialContext method to perform a lookup for a customer EJB home factory. An InitialContext object 214 is produced by the abstract EJB server object 184. The InitialContext object 214 is not generated by the EJB server abstraction layer 134. It is

generated by the concrete implementation of the abstract EJBServer object 184, which in this case is the JbossServer object 186.

- At step 216, the InitialContext object 214 finds the CustomerHome interface 172.

-At step 218, the CustomerMgr object 178 now has a reference to the CustomerHome interface 172, and looks up the CustomerRemote interface 170, by asking the CustomerHome interface 172 to find the customer "Peter" by name.

-At step 220, the CustomerHome interface 172, acts as a factory and instantiates the EJB representation of the customer "Peter" in the EJB container 128. The CustomerHome interface 172 returns the CustomerRemote interface 170, through which all customer "Peter" EJB calls must pass.

-At step 222, the CustomerMgr object 178 then creates a Customer business object, containing a reference to the CustomerRemote interface 170 of the CustomerEJB 120.

-At steps 224-226, the BankTeller object 169 now has a reference to a Customer or the CustomerBean object 174 and is free to complete the deposit transaction by obtaining the customer's or the CustomerBean object's 174 balance and setting the new balance afterwards.

As noted above, in this embodiment, only one line of code needs to change if the business logic 136 is run on

another actual EJB server 132. No other business logic code 136 needs to change.

The following example demonstrates how the single line of code change referred to above can be avoided altogether, so that the business logic code 136 need not change at all to run on another actual EJB server 132.

**Reading from a properties file**

Among other possible methods, the changing of a single line in the business logic 136 can be avoided by saving the type of the actual EJB server 132 to be instantiated in a properties file. Conceptually, this embodiment is the same as the previous example. The only difference is the manner in which the EJB server variable is obtained by the business logic 136.

Instead of instantiating the EJB server reference in the business logic 136 (as was done by the CustomerMgr object 178 in the previous example) it is possible to store this information in a properties file 230, as shown in the block diagram of FIG 10.

The properties file, deployment.properties file 230, acts as the single repository for attributes describing properties to be used by the business logic 136. It is preferable to avoid duplicating logic throughout the business logic 136. By storing all the application attributes in one place, other components or parts of the software code have a convenient place to query application properties. For business components such as the CustomerMgr object 178, the name or type of the actual EJB server 132 is stored in the deployment.properties file 230.

The functionality and Sample Code of the previous example are essentially the same except for the addition of:

- the deployment.properties file 230 which contains the name or type of the actual EJB server 132 to be instantiated at runtime;

- a SiteProperties class 232 that returns the deployment properties specified in the deployment.properties file 230;

- the CustomerMgr class or object 178 also changes slightly. Instead of commenting in the appropriate type of the actual EJB server 132, the deployment.properties file 230 now provides that information. Accordingly, the CustomerMgr class definition changes as shown in the following incomplete Java code.

```
// deployment.properties file
server = JBossServer


// SITEPROPERTIES
public class SiteProperties
{
   Properties _Properties = null;
   EJBServer server = null;
   FileInputStream fis = new FileInputStream(new
File("deployment.properties"));

   static
   {
        // read in the properties file
        _Properties = new Properties("fis");

        // get the name of the concrete server we are going to instantiate
      String serverStr = (String)_Properties.get("server");

        // instantiate the class
      Class EJBServerClass = Class.forName(serverStr);
        server = (EJBServer) EJBServerClass.newInstance();

   }

   public static EJBServer getEJBServer()
   {
        return server;
   }
```

```
}

// CUSTOMER MANAGER
public class CustomerMgr
{

   EJBServer server = null;

   public CustomerMgr()
   {
      // get our EJB server from properties file
      server = SiteProperties.getEJBServer();
   }

   public Customer findCustomer(String name)
   {
      CustomerHome home = (CustomerHome) server.lookupHome("customerhome");
      CustomerRemote remote = (CustomerRemote) home.findName(name);

      Customer customer = new Customer(remote);

      return customer;
   }

}
```

A sequence diagram depicting the order of events for the business logic 136 to use the deployment.properties file 230 at runtime is shown in FIG 11.

- At step 234, the SiteProperties object 232 reads the name or type of actual EJB server 132 to be used from the deployment.properties file 230 as part of the static initialization of the SiteProperties object 232.

- At step 236, the SiteProperties object 232 then creates the JBossServer object 186.

- At step 238, the CustomerMgr object 178 obtains a reference to the EJB server 184 from the SiteProperties object 232.

- At step 240, the CustomerMgr object 178 is able to make and delegate EJB server method calls against the abstract EJBServer interface 184.

The diagrams of FIGs 7 and 8 and sequence diagram of FIG 9 demonstrate how the business logic 136 can be shielded from EJB server object 186-190 details. In this example, a developer does not need to make any changes to the business logic code 136 when porting an EJB application from one actual EJB server 132 to another. All implementation details are hidden behind the abstract EJBServer object 184.

Numerous modifications and variations of the present invention are possible in light of the above teachings. It is therefore to be understood that within the scope of the appended claims, the invention may be practised otherwise than as specifically described herein.

## Appendix "A" To The Detailed Description

### SAMPLE CODE

```
// CLIENT

// BUSINESS LOGIC //

// BANK TELLER
public class BankTeller
{
   public void deposit(float ammount)
   {

      CustomerMgr mgr = new CustomerMgr();
      Customer customer = mgr.findCustomer("Peter");
      float newBalance = ammount + customer.getBalance();
      customer.setBalance(newBalance);
   }
   public static void main(String[] args)
   {

      BankTeller teller = new BankTeller();

      teller.deposit(1000.0f);
   }
}

// CUSTOMER MANAGER
public class CustomerMgr
{

   EJBServer server = null;

   public CustomerMgr()
   {
      // get our EJB server our application is using today
      server = new JBossServer();
      //server = new WebLogicServer();
      //server = new iPlanetServer();
   }

   public Customer findCustomer(String name)
   {
      CustomerHome home = (CustomerHome) server.lookupHome("customerhome");
      CustomerRemote remote = (CustomerRemote) home.findByName(name);

      Customer customer = new Customer(remote);

      return customer;
   }

}
```

```
// CUSTOMER
public class Customer
{
   protected CustomerRemote _ejb = null;

   public Customer(CustomerRemote ejb)
   {
      _ejb = ejb;
   }

   public float getBalance()
   {
       float balance = (float)0.0;
       balance =  _ejb.getBalance();
       return balance;
   }

   public void  setBalance(float balance)
   {
      _ejb.setBalance(balance);
   }

}

// EJB SERVER ABSTRACTION LAYER

public abstract class EJBServer
{

   public Object lookupHome(String jndiName)
   {
      Context ctx = getInitialContext();
      Object home = ctx.lookup(jndiName);
      return home;

   }

   public abstract Context getInitialContext()

   public abstract QueueConnectionFactory getQueueConnectionFactory();

   public abstract TopicConnectionFactory getTopicConnectionFactory();

   public abstract XAConnectionFactory getXAConnectionFactory();

   public abstract XAQueueConnectionFactory getXAQueueConnectionFactory();

   public abstract XATopicConnectionFactory getXATopicConnectionFactory();

   ... any other common services offered by ejb server ...

}

// CONCRETE IMPLEMENTATIONS OF EJB SERVERS

// JBOSS
public class JBossServer extends EJBServer
{

   public Context getInitialContext(String serverUrl) throws NamingException
```

78746-4

```
    {

        Properties prop = new Properties();
        prop.put(Context.INITIAL_CONTEXT_FACTORY,
                "org.jnp.interfaces.NamingContextFactory");
        prop.put(Context.URL_PKG_PREFIXES, "org.jnp.interfaces");
        prop.put(Context.PROVIDER_URL, serverUrl);

        Context context = new InitialContext(prop);

        return context;

    }

    // other potential concrete methods
    public QueueConnectionFactory getQueueConnectionFactory()
    {
        ... etc ...
    }

    public TopicConnectionFactory getTopicConnectionFactory()
    {
        ... etc ...
    }

    public XAConnectionFactory getXAConnectionFactory()
    {
        ... etc ...
    }

    public XAQueueConnectionFactory getXAQueueConnectionFactory()
    {
        ... etc ...
    }

    public XATopicConnectionFactory getXATopicConnectionFactory()
    {
        ... etc ...
    }

}


// WEBLOGIC
public class WebLogicServer extends EJBServer
{

    public Context getInitialContext(String serverUrl) throws NamingException
    {

        Properties prop = new Properties();
        prop.put(Context.INITIAL_CONTEXT_FACTORY,
                "weblogic.jndi.WLInitialContextFactory");
        prop.put(Context.PROVIDER_URL, serverUrl);

        Context context = new InitialContext(prop);

        return context;

    }
```

```
    // other potential concrete methods
    public QueueConnectionFactory getQueueConnectionFactory()
    {
      ... etc ...
    }

    public TopicConnectionFactory getTopicConnectionFactory()
    {
      ... etc ...
    }

    public XAConnectionFactory getXAConnectionFactory()
    {
      ... etc ...
    }

    public XAQueueConnectionFactory getXAQueueConnectionFactory()
    {
      ... etc ...
    }

    public XATopicConnectionFactory getXATopicConnectionFactory()
    {
      ... etc ...
    }


}

// IPLANET
public class iPlanetServer extends EJBServer
{

    public Context getInitialContext(String serverUrl) throws NamingException
    {

      Hashtable env = new Hashtable(1);
      env.put("javax.naming.factory.initial",
              "com.netscape.server.jndi.RootContextFactory");
      Context context = new InitialContext(env);

      return context;
    }

    // other potential concrete methods
    public QueueConnectionFactory getQueueConnectionFactory()
    {
      ... etc ...
    }

    public TopicConnectionFactory getTopicConnectionFactory()
    {
      ... etc ...
    }

    public XAConnectionFactory getXAConnectionFactory()
    {
      ... etc ...
    }

    public XAQueueConnectionFactory getXAQueueConnectionFactory()
    {
```

```
      ... etc ...
   }

   public XATopicConnectionFactory getXATopicConnectionFactory()
   {
      ... etc ...
   }


}


// SERVER

// CUSTOMER EJB

// REMOTE INTERFACE
public interface CustomerRemote extends EJBObject
{
   public float getBalance() throws RemoteException;
   public void setBalance(float balance)  throws RemoteException;
}

// HOME INTERFACE
public interface CustomerHome extends EJBHome
{

   public CustomerRemote create(String CustomerId,
                                String name,
                                String password,
                                float initialBalance)
     throws CreateException, RemoteException;

  public CustomerRemote findByPrimaryKey(CustomerPK primaryKey)
     throws FinderException, RemoteException;

  public CustomerRemote findByName(String name)
     throws FinderException, RemoteException;

}

// ENTERPRISE JAVA BEAN
public class CustomerBean implements EntityBean
{

   protected EntityContext ctx;
   protected String _customerId; // also the primary Key
   protected String _name;
   protected String _password;
   protected float _balance;


   public float getBalance()
   {
      return _balance;
   }

   public void setBalance(float newBalance)
   {
      _balance = newBalance;
   }
```

```
    public CustomerPK ejbCreate(String customerId,
                                String name,
                                String password,
                                float initialBalance)
       throws CreateException, RemoteException
    {
       _balance = initialBalance;
    }

    public CustomerPK ejbFindByPrimaryKey(CustomerPK pk)
       throws FinderException, RemoteException
    {}

    public CustomerPK ejbFindByName(String name)
       throws FinderException, RemoteException
    {}

    public void ejbStore() throws RemoteException
    {}

    public void ejbRemove() throws RemoveException, RemoteException
    {}

    public void ejbLoad()
     throws RemoteException
    {}

    public void ejbPostCreate(String customerId,
                              String name,
                              String password,
                              float initialBalance)
    {}

    public void ejbActivate()
    {}

    public void setEntityContext(EntityContext ctx)
     throws RemoteException
    {}

    public void unsetEntityContext()
     throws RemoteException
    {}

    public void ejbPassivate()
    {}
}
```

78746-4

What we claim as our invention is:

1. A computer-readable medium containing executable instructions for abstracting away differences among two or more distributed object oriented structure (DOOS) servers, the instructions being adapted to be used in combination with business logic software code for communicating with a DOOS, a DOOS having a home interface, a remote interface and an implementation class, wherein the instructions when loaded in a computer, adapt the computer to receive the identity of a current DOOS server from the business logic software code, and enable the business logic software to communicate with a DOOS moved from a previous DOOS server to a current DOOS server.

2. The computer-readable medium of claim 1, wherein the two or more DOOS servers support a number of services in common, wherein the instructions comprise methods associated with services in common.

3. The computer-readable medium of claim 2 wherein the instructions define an object oriented abstract class and for each of the DOOS servers, the instructions further define a corresponding object oriented concrete class.

4. The computer-readable medium of claim 3 wherein the abstract class comprises an abstract method for each of the services in common offered by the two or more DOOS servers and each of the concrete classes comprises a number of methods wherein for each abstract method of the abstract class, there is a corresponding concrete method in each of the concrete classes, each concrete method being for implementing a service on a corresponding DOOS server.

5. The computer-readable medium of claim 4 wherein the DOOS is an Enterprise JavaBean.

6. The computer-readable medium of claim 5, the instructions being for abstracting away differences relating to how different DOOS servers use one or more Java 2 Enterprise Edition platform features.

7. The computer-readable medium of claim 6 wherein the one or more Java 2 Enterprise Edition platform features comprise one or more of a Java Messaging Service, a Java Naming and Directory Interface, a Java Database Connectivity feature and a Java Transaction API and Service (JTS/JTS).

8. A computer-readable medium storing computer software and data that when loaded by a computing device define object oriented objects for use in abstracting away differences among two or more distributed object oriented structure (DOOS) servers, the two or more DOOS servers supporting a number of common service, a DOOS having a home interface, a remote interface, and an implementation class, the computer software and data being implementation by the computing device in an object oriented framework comprising:

-a first object comprising an abstract class having, for each of the common services, a corresponding abstract method; and

-for each of the DOOS servers, an associated object comprising a concrete sub-class of the abstract class of the first object, each concrete sub-class comprising, for each of the abstract methods of the abstract class, an associated concrete method, wherein each concrete method is adapted to implement a service on the associated DOOS server.

9. The computer-readable medium of claim 8 wherein the first object and the second object comprise methods for abstracting away differences relating to how different DOOS

servers use one or more Java 2 Enterprise Edition platform features.

10. The computer-readable medium of claim 9 wherein the one or more Java 2 Enterprise Edition platform features comprise one or more of a Java Messaging Service, a Java Naming and Directory Interface, a Java Database Connectivity feature and a Java Transaction API and Service (JTS/JTS).

11. A computer implemented method for abstracting away differences among two or more distributed object oriented structure (DOOS) servers, a DOOS having a home interface, a remote interface and an implementation class, the DOOS servers supporting a number of common services, the method comprising implementing business logic and a DOOS server abstraction layer wherein:

the business logic:

(a) provides an identity of a DOOS server incorporating one or more DOOSs;

(b) obtains, through the DOOS server abstraction layer, for each of the one or more DOOSs, an associated home interface;

(c) obtains, for each of the one or more DOOSs, an associated remote interface;

(d) for each DOOS, uses the associated home interface and the associated remote interface to communicate with the DOOS through the DOOS abstraction layer, using non-DOOS server specific abstract methods;

the DOOS server abstraction layer:

(e) obtains the home interface for each of the one or more DOOSs for the business logic; and

(f) associates each non-DOOS server specific abstract method with a DOOS server specific concrete method.

12. The method of claim 11 wherein the DOOS abstraction layer comprises

-an object oriented abstract class having a non-DOOS server specific abstract method for each of the common services offered by the two or more DOOS servers;

-for each of the DOOS servers, a corresponding object oriented concrete sub-class of the abstract class, each of the concrete sub-classes comprising a number of DOOS server specific concrete methods wherein for each of the non-DOOS server specific abstract methods of the abstract class, there is a corresponding DOOS server specific concrete method in each of the concrete sub-classes, each DOOS server specific concrete method being for implementing a service on a corresponding DOOS server,

wherein step (f) comprises associating a non-DOOS server specific abstract method of the abstract class with a corresponding DOOS server specific concrete method of a concrete sub-class.

13. The method of claim 12 wherein step (a) comprises hard coding the identity of the DOOS server.

14. The method of claim 12 wherein step (a) comprises obtaining the identity of the DOOS server from a file.

15. The computer-readable medium of claim 12 wherein the DOOS is an Enterprise JavaBean.

78746-4

16. The computer-readable medium of claim 15, the instructions being for abstracting away differences relating to how different DOOS servers use one or more Java 2 Enterprise Edition platform features.

17. The computer-readable medium of claim 16 wherein the one or more Java 2 Enterprise Edition platform features comprise one or more of a Java Messaging Service, a Java Naming and Directory Interface, a Java Database Connectivity feature and a Java Transaction API and Service (JTS/JTS).

SMART & BIGGAR
OTTAWA, CANADA

PATENT AGENTS

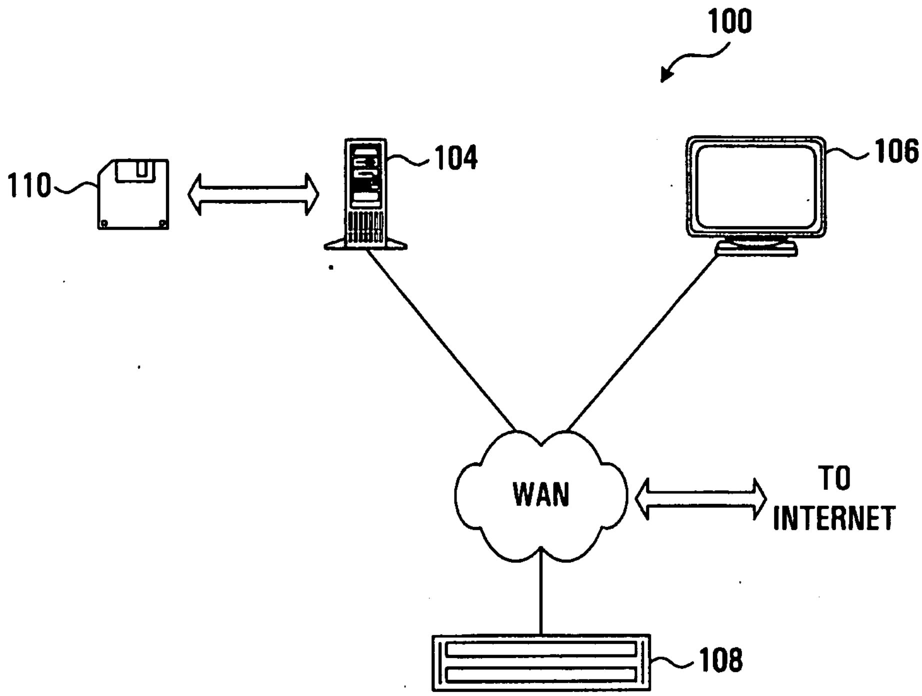

100
110
104
106
WAN
TO
INTERNET
108

FIG. 1

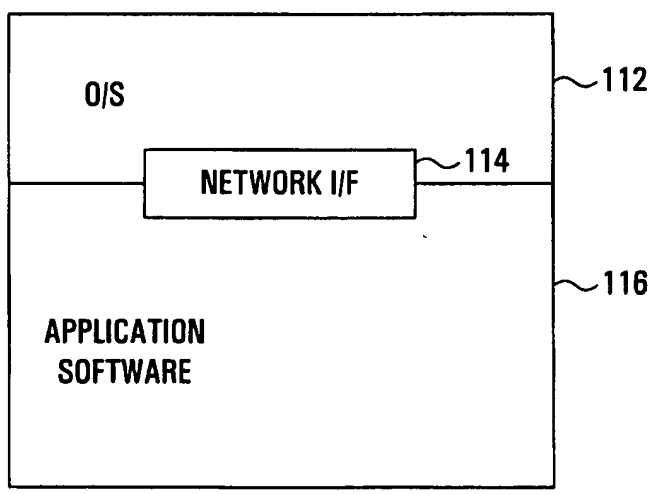

O/S
112
NETWORK I/F
114
116
APPLICATION
SOFTWARE

FIG. 2

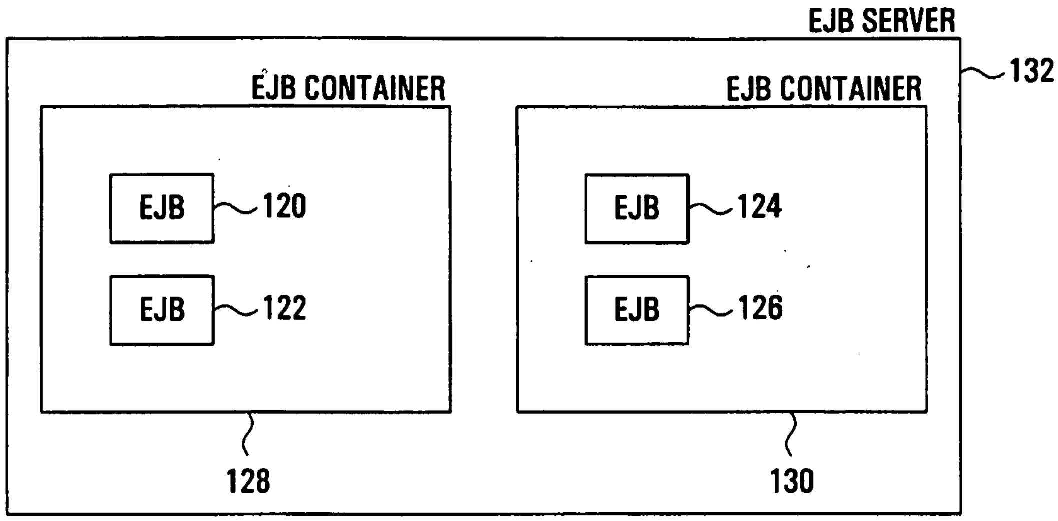

EJB SERVER
EJB CONTAINER
EJB CONTAINER
EJB
EJB
EJB
EJB
120
122
124
126
128
130
132

FIG. 3

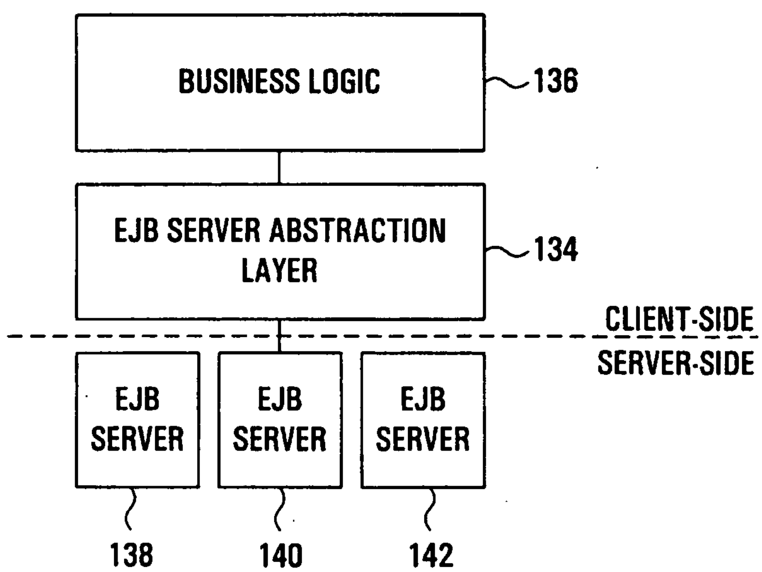

BUSINESS LOGIC
136
EJB SERVER ABSTRACTION LAYER
134
CLIENT-SIDE
SERVER-SIDE
EJB SERVER
EJB SERVER
EJB SERVER
138
140
142

FIG. 4

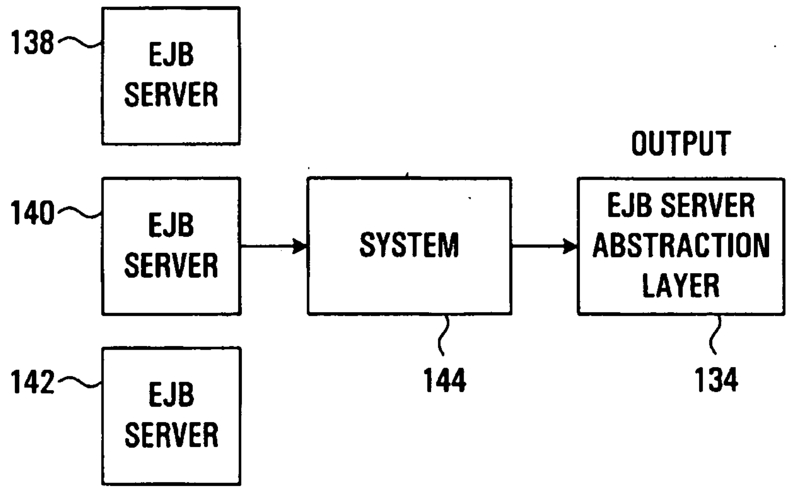

138
EJB SERVER
140
EJB SERVER
142
EJB SERVER
SYSTEM
144
OUTPUT
EJB SERVER ABSTRACTION LAYER
134

FIG. 5

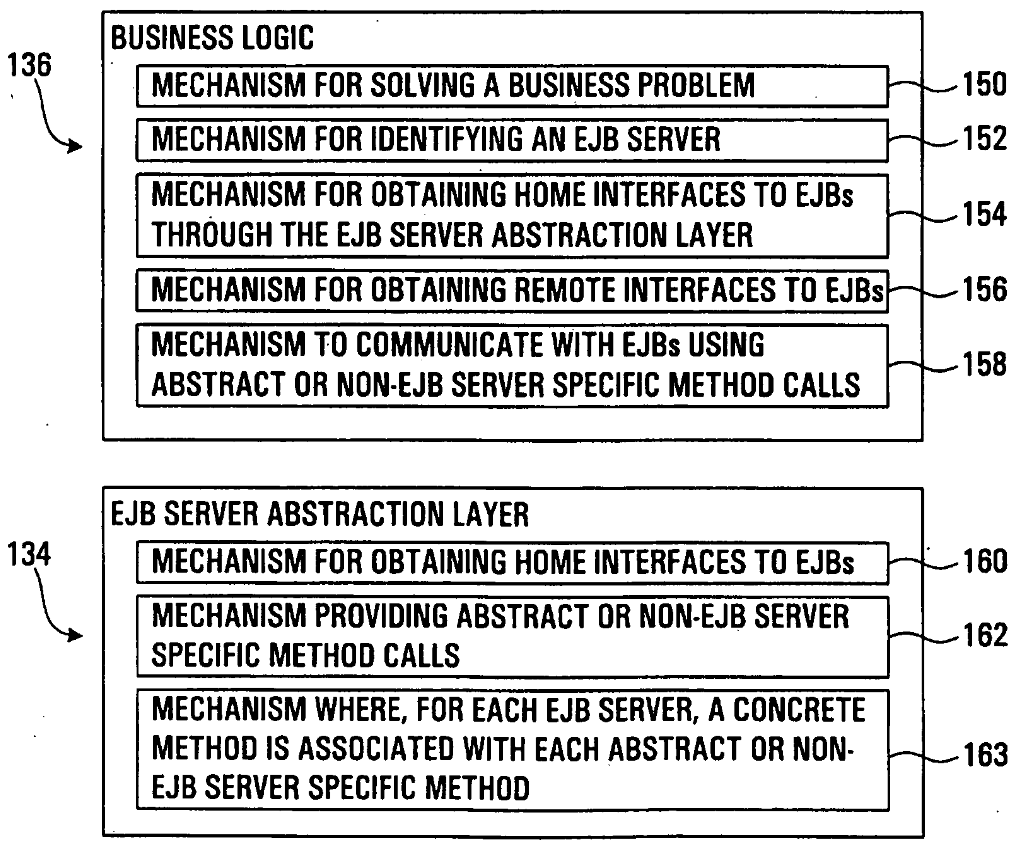

136
BUSINESS LOGIC
MECHANISM FOR SOLVING A BUSINESS PROBLEM
150
MECHANISM FOR IDENTIFYING AN EJB SERVER
152
MECHANISM FOR OBTAINING HOME INTERFACES TO EJBs THROUGH THE EJB SERVER ABSTRACTION LAYER
154
MECHANISM FOR OBTAINING REMOTE INTERFACES TO EJBs
156
MECHANISM TO COMMUNICATE WITH EJBs USING ABSTRACT OR NON-EJB SERVER SPECIFIC METHOD CALLS
158
134
EJB SERVER ABSTRACTION LAYER
MECHANISM FOR OBTAINING HOME INTERFACES TO EJBs
160
MECHANISM PROVIDING ABSTRACT OR NON-EJB SERVER SPECIFIC METHOD CALLS
162
MECHANISM WHERE, FOR EACH EJB SERVER, A CONCRETE METHOD IS ASSOCIATED WITH EACH ABSTRACT OR NON-EJB SERVER SPECIFIC METHOD
163

FIG. 6

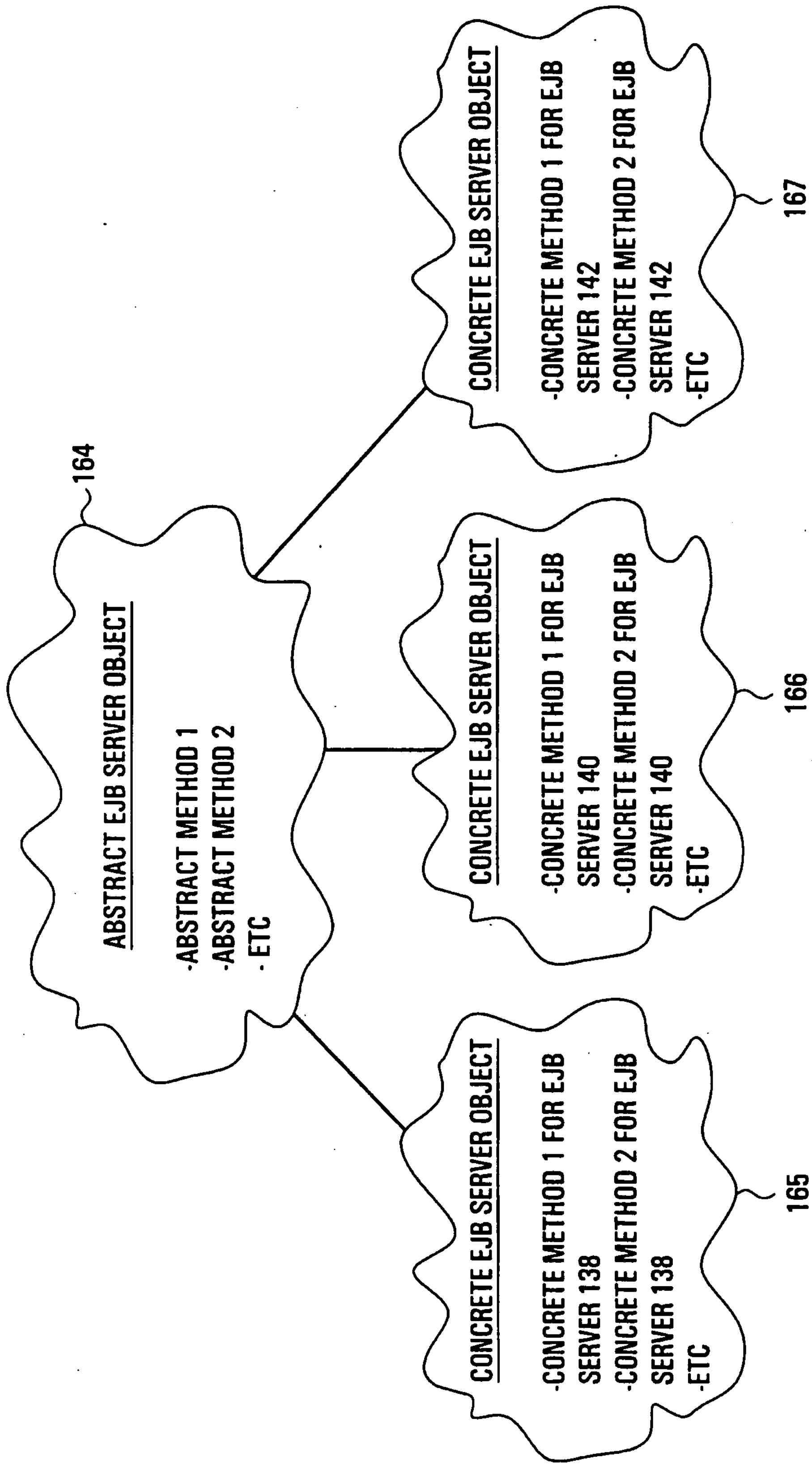

ABSTRACT EJB SERVER OBJECT
-ABSTRACT METHOD 1
-ABSTRACT METHOD 2
- ETC
164
CONCRETE EJB SERVER OBJECT
-CONCRETE METHOD 1 FOR EJB SERVER 138
-CONCRETE METHOD 2 FOR EJB SERVER 138
-ETC
165
CONCRETE EJB SERVER OBJECT
-CONCRETE METHOD 1 FOR EJB SERVER 140
-CONCRETE METHOD 2 FOR EJB SERVER 140
-ETC
166
CONCRETE EJB SERVER OBJECT
-CONCRETE METHOD 1 FOR EJB SERVER 142
-CONCRETE METHOD 2 FOR EJB SERVER 142
-ETC
167

FIG. 7

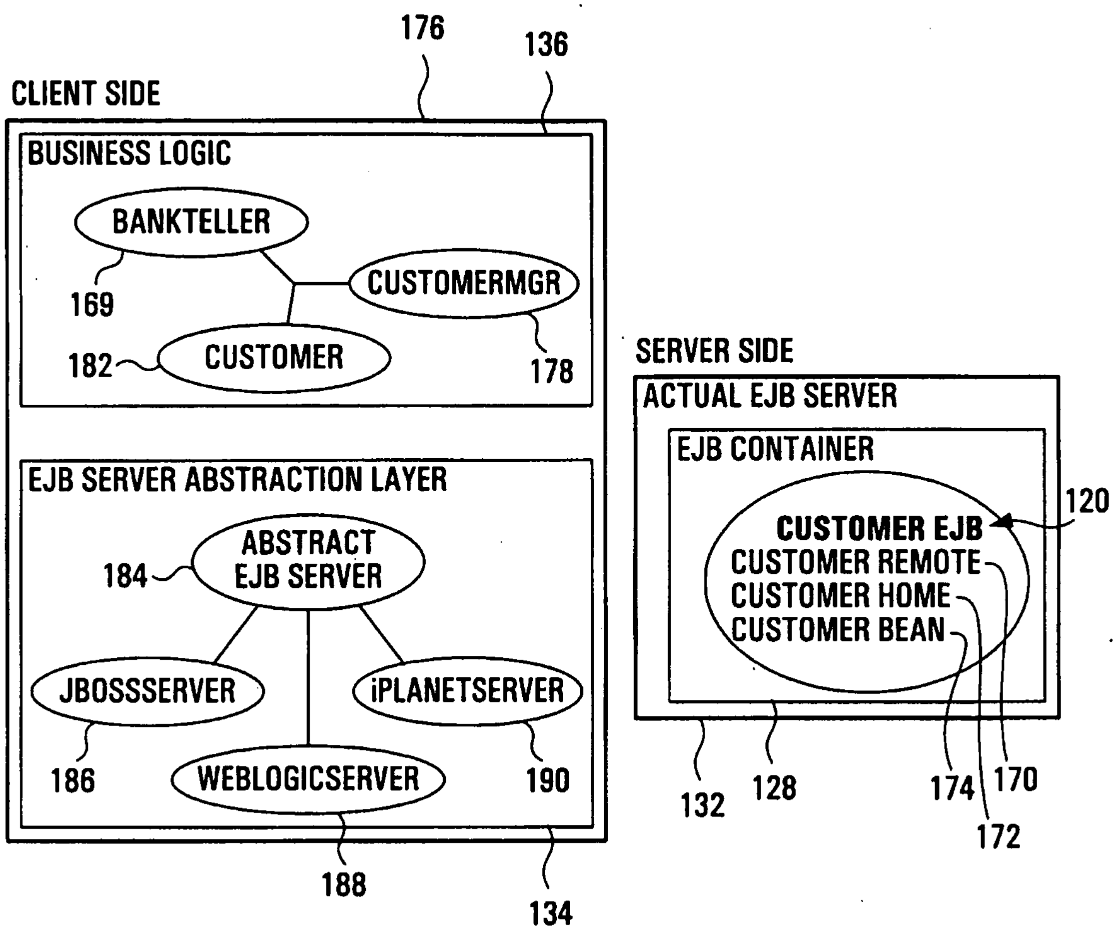

176
136
CLIENT SIDE
BUSINESS LOGIC
BANKTELLER
CUSTOMERMGR
169
CUSTOMER
182
178
EJB SERVER ABSTRACTION LAYER
ABSTRACT
EJB SERVER
184
JBOSSSERVER
iPLANETSERVER
WEBLOGICSERVER
186
190
188
134
SERVER SIDE
ACTUAL EJB SERVER
EJB CONTAINER
CUSTOMER EJB
CUSTOMER REMOTE
CUSTOMER HOME
CUSTOMER BEAN
120
132
128
174
170
172

FIG. 8

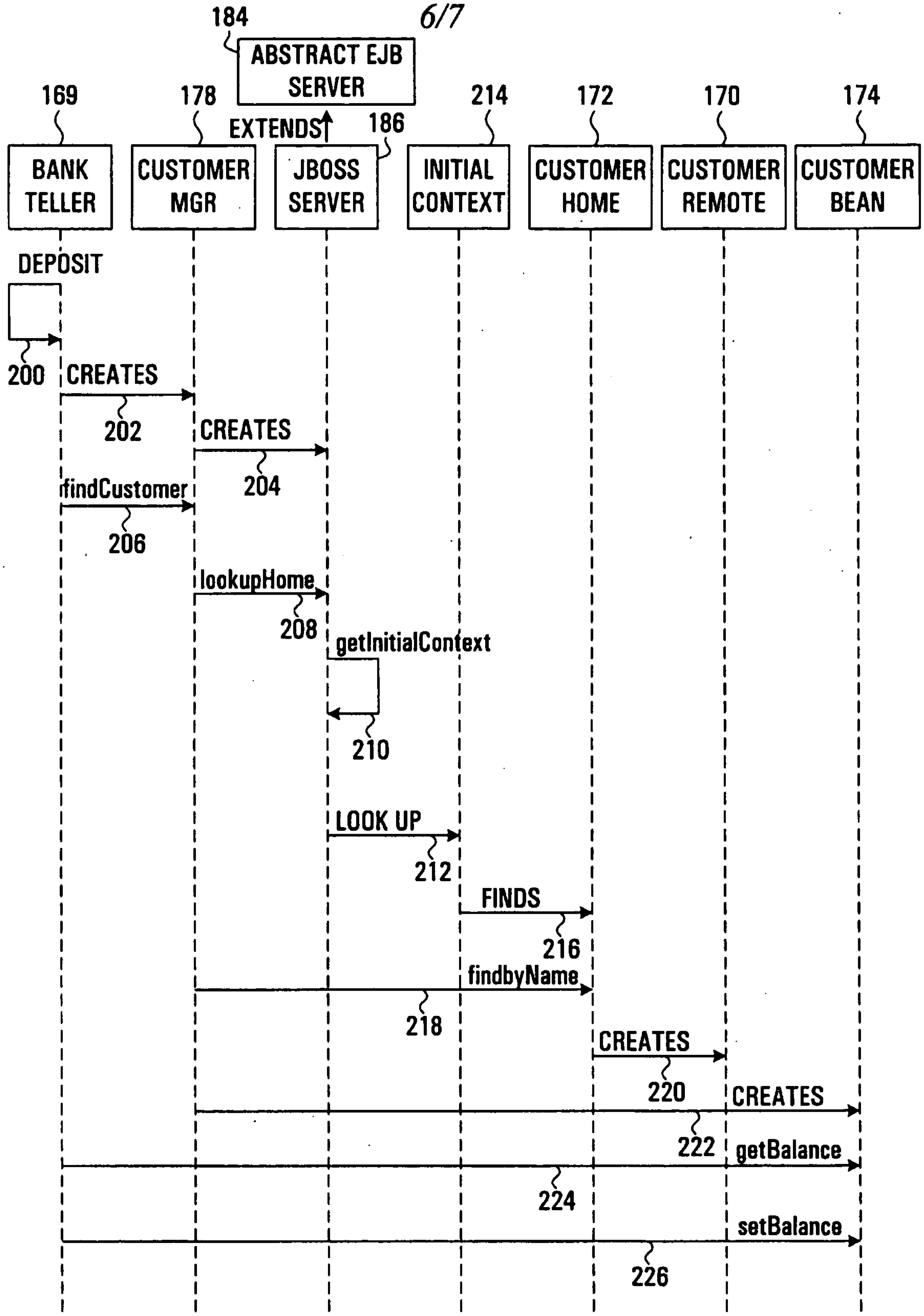

184
ABSTRACT EJB SERVER
EXTENDS
186
169
178
214
172
170
174
BANK TELLER
CUSTOMER MGR
JBOSS SERVER
INITIAL CONTEXT
CUSTOMER HOME
CUSTOMER REMOTE
CUSTOMER BEAN
DEPOSIT
200
CREATES
202
CREATES
204
findCustomer
206
lookupHome
208
getInitialContext
210
LOOK UP
212
FINDS
216
findbyName
218
CREATES
220
CREATES
222
getBalance
224
setBalance
226

FIG. 9

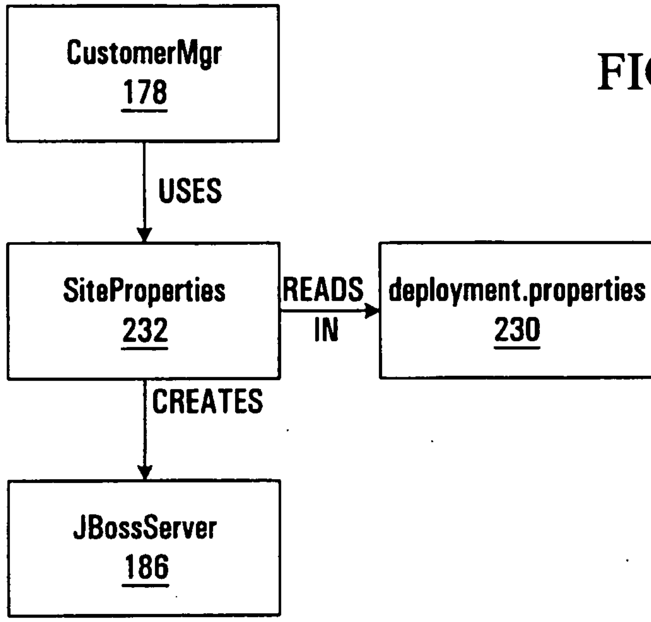

CustomerMgr
178
USES
SiteProperties
232
READS IN
deployment.properties
230
CREATES
JBossServer
186

FIG. 10

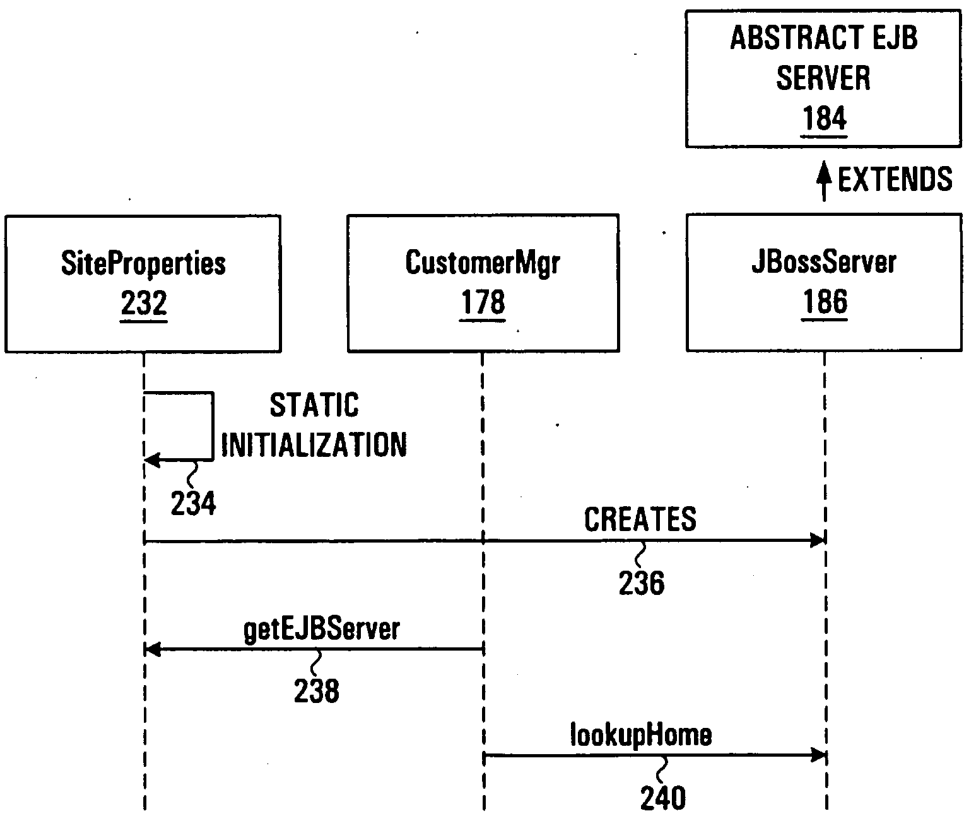
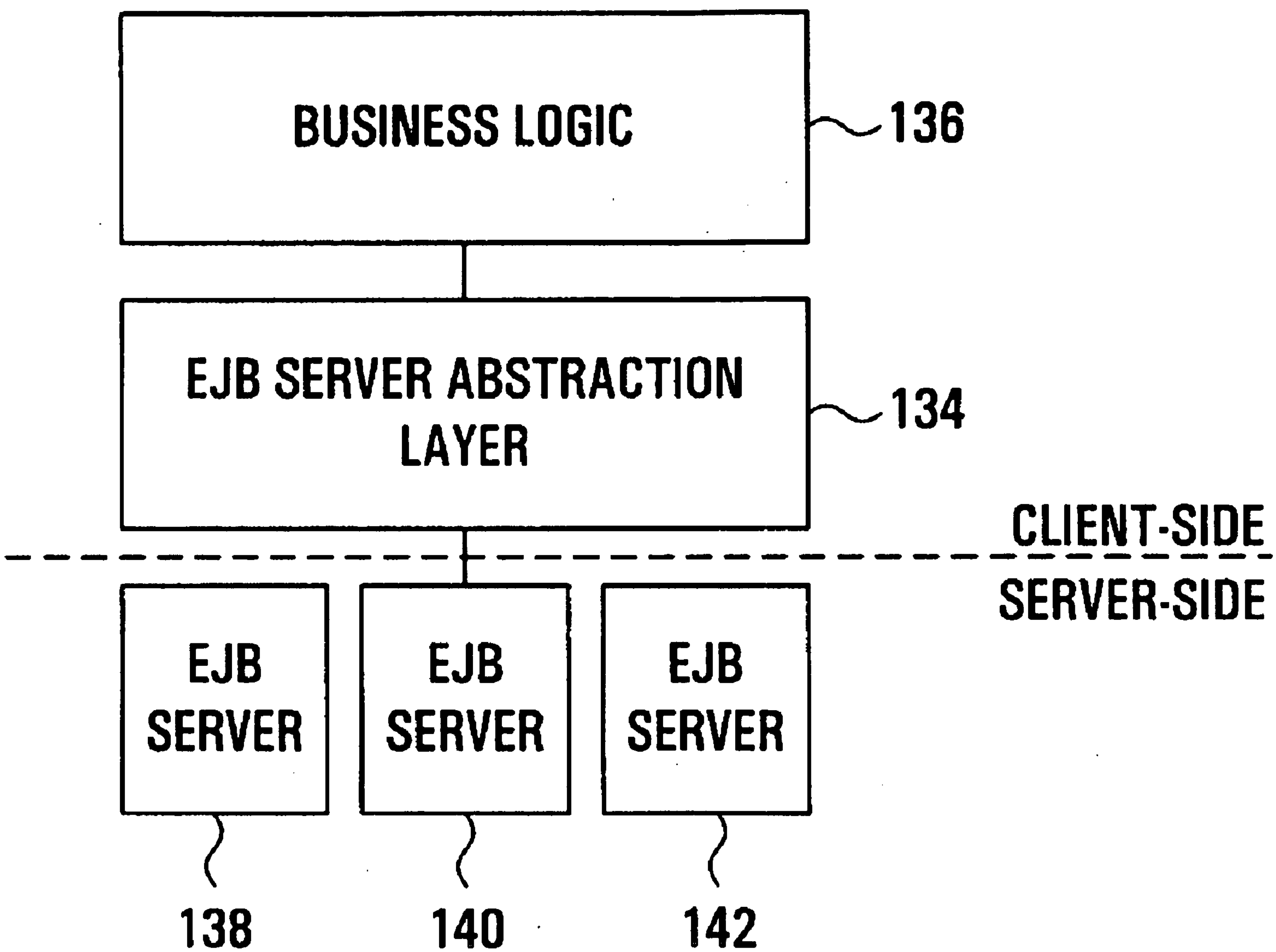

ABSTRACT EJB SERVER
184
EXTENDS
SiteProperties
232
CustomerMgr
178
JBossServer
186
STATIC INITIALIZATION
234
CREATES
236
getEJBServer
238
lookupHome
240

FIG. 11


BUSINESS LOGIC
136
EJB SERVER ABSTRACTION LAYER
134
CLIENT-SIDE
SERVER-SIDE
EJB SERVER
EJB SERVER
EJB SERVER
138
140
142